Programming in Haskell – Homework Assignment 6

UNIZG FER, 2014/2015

Handed out: November 9, 2014. Due: November 16, 2014 at 23:59

Note: Define each function with the exact name and the type specified. You can (and in most cases you should) define each function using a number of simpler functions. Provide a type signature above each function definition and comment the function above the type signature. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message. Problems marked with a star ($\star$) are optional.

Each problem is worth a certain number of points. The points are given at the beginning of each problem or subproblem (if they are scored independently). These points are scaled, together with a score for the in-class exercises, if any, to 10. Problems marked with a star ($\star$) are scored on top of the mandatory problems, before scaling. The score is capped at 10, but this allows for a perfect score even with problems remaining unsolved

1. *(1 pt)* Define a function `partition` to partition a list using a user-provided predicate function.

   ```
   partition :: [a -> Bool] -> [a] -> [[a]]
   partition [odd, even, const True] [1..10]
   ⇒ [[1,3,5,7,9],[2,4,6,8,10],[1,2,3,4,5,6,7,8,9,10]]
   partition [isUpper, isDigit] "Catch 22" ⇒ ["C","22"]
   partition [] "Whatever" ⇒ []
   ```

2. *(1 pt)* Define a function `cycleMap fs xs` that maps various functions from `fs` over a list `xs`, depending on the index of an element in the list. The list `fs` of functions to be mapped is cycled over the list `xs`: the first function from `fs` is applied on the first element from `xs`, the second function from `fs` on the second element from `xs`, etc. When the list of functions `fs` is exhausted, mapping restarts from the first function from `fs`.

   ```
   cycleMap :: [a -> b] -> [a] -> [b]
   cycleMap [odd, even] [1,2,3,4] ⇒ [True, True, True, True]
   cycleMap [(+1), (subtract 1)] [1..10] ⇒ [2,1,4,3,6,5,8,7,10,9]
   cycleMap [map (+1),map (`div` 2),filter (>7)] [[1,2,3],[4,5,6],[7,8,9]]
   ⇒ [[2,3,4],[2,2,3],[8,9]]
   cycleMap [] "Whatever" ⇒ []
   ```

3. *(1 pt)*

   (a) Define an explicitly recursive function `reduce` that reduces a list of elements to a single element using a seed value and a binary reduction function. The reduction function is applied to the seed value and the first element of the list to get an intermediate value. That value is then combined with the second

element of the list to get the next intermediate value, and so on, until the end of the list is reached.

```
reduce :: (a -> b -> a) -> a -> [b] -> a
reduce (+) 0 [1,2,3] ⇒ 6
reduce (-) 0 [1,2,3] ⇒ -6
reduce (++) "a" ["b", "c"] ⇒ "abc"
reduce (\x s -> length s + x) 0 ["an", "example", "or", "something"]
⇒ 20
reduce (*) 1 [] ⇒ 1
```

(b) Define a variant of `reduce` called `reduce1` that behaves like `reduce`, but assumes the input list contains at least one element and so eschews taking a seed element.

```
reduce1 :: (a -> a -> a) -> [a] -> a
reduce1 (+) [1..3] ⇒ 6
reduce1 (++) $ words "just an example" ⇒ "justanexample"
reduce1 (-) [7] ⇒ 7
reduce1 (*) [] ⇒ error "reduce1 got an empty list"
```

(c) Define a function `scan` that performs similarly to `reduce`, but returns a list of all the intermediate values with the result at the end instead of just the last result.

```
scan :: (a -> b -> a) -> a -> [b] -> [a]
scan (+) 0 [1,2,3] ⇒ [0,1,3,6]
scan (-) 0 [1,2,3] ⇒ [0,-1,-3,-6]
scan (*) 0 [] ⇒ [0]
```

4. *(1 pt)*

(a) Define a variant of `reduce` that performs similarly, only does the operations from right to left, instead. Call this function `rreduce`.

```
rreduce :: (a -> b -> b) -> b -> [a] -> b
rreduce (+) 0 [1,2,3] ⇒ 6
rreduce (-) 0 [1,2,3] ⇒ 2
rreduce (++) "a" ["b", "c"] ⇒ "bca"
rreduce (\s x -> length s + x) 0 ["an", "example", "or", "something"]
⇒ 20
rreduce (*) 1 [] ⇒ 1
```

(b) Define a variant of `rreduce` called `rreduce1` that behaves like `rreduce`, but assumes the input list contains at least one element.

```
rreduce1 :: (a -> a -> a) -> [a] -> a
rreduce1 (+) [1..3] ⇒ 6
rreduce1 (++) $ words "just an example" ⇒ "justanexample"
rreduce1 (-) [7] ⇒ 7
rreduce1 (ʃ) [] ⇒ error "rreduce1 got an empty list"
```

(c) Define a variant of the `scan` function that works from right to left, called `rscan`.

```
rscan :: (a -> b -> b) -> b -> [a] -> [b]
rscan (+) 0 [1,2,3] ⇒ [6, 5, 3, 0]
rscan (-) 0 [1,2,3] ⇒ [2,-1,3,0]
rscan (*) 0 [] ⇒ [0]
```

5. *(1pt)*

   (a) Define `newton`, a function that computes an approximation of the square root of a number using a special case of Newton's method. Assuming that some initial guess `y` is the square root of `x`, we can get a better approximation of the actual square root (`y'`) by averaging `y` and `x/y`. In other words, `y' = (y+x/y)/2`. We repeat this step with the newly found value `y'` to gain better and better approximations. The function should return a result when the difference between two successive approximations is less than some given tolerance. (*Note:* Your results might differ from the ones shown below, but should in general show improvements as the tolerance value gets smaller.)

```
type Tolerance = Double
newton :: Tolerance -> Double -> Double
newton 1e+1 1024 ⇒ 32.02142090500024
newton 1e-2 1024 ⇒ 32.0000071648159
newton 1e-4 1024 ⇒ 32.0000000000008
newton 1e-8 0 ⇒ 6.103515625e-5
newton any (-632) ⇒ error "can't get sqrt of negative number"
```

   (b) Define `deriv`, a function that computes the derivative of a given function. Remember that the derivative of `f(x)` is defined as `f'(x) = [f(x+dx)-f(x)]/dx`. We'll cheat and only return an approximation of the derivative: simply assume that dx is equal to a very small number (like `0.00001`). (*Note:* Again, your results might differ slightly.)

```
deriv :: (Double -> Double) -> Double -> Double
let f = (**3)
let f' = deriv f
f' 2 ⇒ 12.000060000261213
let g' = deriv sin
g' 3.14159 ⇒ -0.9999999999996315
```

6. *(1 pt)* Happiness is important. Define a function `isHappy` that checks whether a number is a happy number. The process of checking goes like this: replace a number with the sum of the squares of its digits. If the new number equals `1`, the number is *happy*. Otherwise, if we have reached a number previously calculated in the same cycle, the number is *unhappy* (or *sad*). You may use a list to keep track of numbers that previously appeared in the cycle. However, this is inefficient and you may prefer to (but do not have to) use `Data.Set` instead. You can use the `Data.Char.digitToInt` function to extract digits from a number. Once you've implemented this function, you can use it to shun sad numbers.

```
isHappy :: Int -> Bool
isHappy 1 ⇒ True
isHappy 5 ⇒ False
isHappy 28 ⇒ True
take 5 $ filter isHappy [1..] ⇒ [1,7,10,13,19]
```

7. *(2 pts)*

   (a) Define a function `split` which recursively splits a list into two sublists at the middle. The time complexity of this function must be $\mathcal{O}(n)$.

```
split :: [a] -> ([a], [a])
split [1..4] ⇒ ([1,2], [3,4])
split [1..5] ⇒ ([1,2,3], [4,5])
split [] ⇒ ([], [])
```

(b) Using the `split` function you just wrote, implement `mergesort`, a function which performs the mergesort sorting algorithm on a list.

```
mergesort :: Ord a => [a] -> [a]
mergesort xs == sort xs
```

(c) Define the function `countInversions` which counts how many swaps of adjacent elements have to be performed to sort a list. Implement it using a modified version of the `mergesort` function.

```
countInversions :: Ord a => [a] -> Int
countInversions [] ⇒ 0
countInversions [1..5] ⇒ 0
countInversions $ reverse [1..5] ⇒ 10
countInversions [100,50,75,25] ⇒ 5
```

(d) Define a second version of mergesort, called `mergesort'`. It should operate on `Data.Vector` instead.

```
mergesort' :: Ord a => Vector a -> Vector a
mergesort' v == fromList $ sort $ toList v
```

If you do not have the `Data.Vector` package installed, you can install it using cabal by simply typing in:

```
$ cabal install vector
```

8. *(1.5 pts)* Suppose you have a bundle of `n` packages (numbered from 1 to `n`) that need to be installed. It is possible that some packages depend on some other packages (which can also have their own dependencies). Your job is to sort this mess out by defining a function `pacMan` that takes a number `n` and a list of dependencies and returns a list of `n` elements describing the order in which packages should be installed.

   Clarifications:

   - Package `A` depends on package `B` if `B` needs to be installed before `A`;
   - Dependencies are represented as tuples where `(A,B)` means that `A` depends on `B`;
   - If there are multiple solutions, any of them is considered valid.

```
type Package = Int

-- | Package A depends on Package B in (A,B)
type Dependency = (Package,Package)

pacMan ::  Int -> [Dependency] -> [Package]
pacMan 3 [(1,2),(2,3),(3,21)] ⇒ error "Impossible to resolve"
pacMan 3 [(1,2),(2,3)] ⇒ [3,2,1]2

-- | Following two examples have multiple valid solutions
pacMan 4 [(1,3),(1,2),(2,4)] ⇒ [4,2,3,1]
pacMan 5 [(1,2),(2,3),(5,4)] ⇒ [4,3,5,2,1]
```

9.⋆ *(2 pts)* In this task, you will create life. You are to implement Conway's Game of Life, a simple cellular automaton originally imagined by the mathematician John Horton Conway. Please follow the given link to familiarise yourself with the rules.

We'll represent the state of the world with a `Data.Vector`-based matrix of booleans. A cell with the value `True` represents a living cell, while a cell with the value `False` represents a dead cell.

```
type World = Vector (Vector Bool)
miniworld = fromList $ map fromList [[True, True, False], [False, True,
True], [False, True, False]]
```

(a) You will need to create your new world. Define a function `genesis` that takes a width and a height and constructs a new random `World`.

```
genesis ::  Int -> Int -> IO World
```

The function should throw an error when either dimension is less than `1`. You can use the following function to generate random values:

```
randBools :: Int -> IO [Bool]
randBools len howmuch = do
  g <- newStdGen
  let bools = randoms g :: [Bool]
  return $ take n bools
```

(b) Implement a function `showWorld` that displays your world. Living cells are marked with the character `'x'` while dead cells are marked with a space.

```
showWorld :: World -> String showWord miniworld == "xx \n xx\n x "
```

(c) To implement the Game of Life, we first need to know the number of neighbours a certain cell has. Implement a function `neighbours` that takes a world and the row and column indices (`0`-indexed) and returns the number of living neighbours the cell has (out of a maximum of eight).

```
neighbours :: World -> Int -> Int -> Int
neighbours miniword 0 0 ⇒ 2
neighbours miniworld 2 0 ⇒ 2
neighbours miniworld 1 1 ⇒ 4
```

(d) We need to know whether a cell at a certain position is going to be alive or dead in the next step. Define a function `nextState` that takes a world, a row index and a column index and returns whether that cell will be alive in the next game step.

```
nextState :: World -> Int -> Int -> Bool
nextState miniworld 0 0 ⇒ True
nextState miniworld 0 2 ⇒ True
nextState miniworld 1 1 ⇒ False
```

(e) Now define a function `gameStep` that takes an instance of the world and returns the world in the next step.

```
gameStep :: World -> World
showWorld $ gameStep miniworld == "xxx\n  x\n xx"
```

(f) Define a function `play` that takes an initial, seed world and then repeatedly does the following: displays the world, calculates the next state of the world and waits for one second. Using this function, you can see your world in action.

(Hint: You can implement waiting with `Control.Concurrent.threadDelay` – note that it takes microseconds as the argument).

```
play :: World -> IO ()
```

(g) Finally, define a `main` function that takes the arguments – the width and the height of the board – from the command line and then constructs and plays the world on screen using the `play` function. You have mastered the power of creation. Good job!

Corrections

Revision 1.1 – Fixed type signatures for `reduce1` and `rreduce1`
Revision 1.2 – Corrected return value for `cycleMap`
Revision 1.3 – Fixed incorrect example for `gameStep`